CLASSES, INHERITANCE, EXCEPTIONS

Class Fundamentals

- A class is a *template* for an object, and an object is an *instance* of a class

The General Form of a Class

class *classname* {
type instance-variable1;
type instance-variable2;
// ...
type instance-variableN;
type methodname1(*parameter-list*) {
// body of method
}
type methodname2(p*arameter-list*) {
// body of method
}
// ...
type methodnameN(*parameter-list*) {
// body of method
} }

- Collectively, the methods and variables defined within a class are called *members* of the class
- Variables defined within a class are called instance variables because each instance of the class contains its own copy of these variables

A Simple Class

```
class Box {
double width;
double height;
double depth;
}
Box mybox = new Box();
// create a Box object called mybox
```

- To access these variables, you will use the *dot* (.) operator
- The dot operator links the name of the object with the name of an instance variable

Example: mybox.width = 100;

```
class Box {
double width;
double height;
double depth;
}
// This class declares an object of type Box.
class BoxDemo {
public static void main(String args[]) {
Box mybox = new Box();
double vol;
// assign values to mybox's instance variables
mybox.width = 10;
mybox.height = 20;
mybox.depth = 15;
```

```
// compute volume of box
vol = mybox.width * mybox.height * mybox.depth;
System.out.println("Volume is " + vol);
} }
```

- When you compile this program, you will find that two **.class** files have been created, one for **Box** and one for **BoxDemo**
- The Java compiler automatically puts each class into its own **.class** file

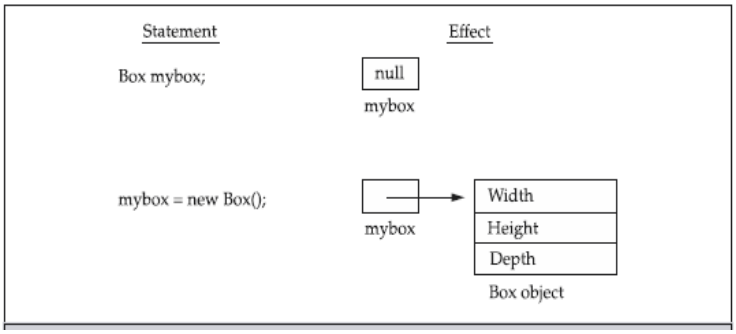
Declaring Objects

- First, you must declare a variable of the class type
- This variable does not define an object ; Instead, it is simply a variable that can *refer* to an object
- Second, you must acquire an actual, physical copy of the object and assign it to that variable. You can do this using the **new** operator
- The **new** operator dynamically allocates memory for an object and returns a reference to it

```
Box mybox; // declare reference to object
mybox = new Box(); // allocate a Box object
```

OR

```
Box mybox = new Box();
```



Assigning Object Reference Variables

Example:

```
Box b1 = new Box();
Box b2 = b1;
```

- **b1** and **b2** will both refer to the *same* object
- The assignment of **b1** to **b2** did not allocate any memory or copy any part of the original object
- Any changes made to the object through **b2** will affect the object to which **b1** is referring, since they are the same object
- Although **b1** and **b2** both refer to the same object, they are not linked in any other way

Example:

```
Box b1 = new Box();
Box b2 = b1;
// ...
```

```
b1 = null;
```

- Here, **b1** has been set to **null**, but **b2** still points to the original object

Introducing Methods

type name(*parameter-list*) {
// body of method
}

- *type* specifies the type of data returned by the method
- If the method does not return a value, its return type must be **void**
- The *parameter-list* is a sequence of type and identifier pairs separated by commas

```
class Box {
double width;
double height;
double depth;
// compute and return volume
double volume() {
return width * height * depth;
}
// sets dimensions of box
void setDim(double w, double h, double d) {
width = w;
height = h;
depth = d;
} }
class BoxDemo5 {
public static void main(String args[]) {
Box mybox1 = new Box();
Box mybox2 = new Box();
double vol;
// initialize each box
mybox1.setDim(10, 20, 15);
mybox2.setDim(3, 6, 9);
// get volume of first box
vol = mybox1.volume();
System.out.println("Volume is " + vol);
// get volume of second box
vol = mybox2.volume();
System.out.println("Volume is " + vol);
} }
```

Constructors

- A *constructor* initializes an object immediately upon creation
- It has the same name as the class in which it resides and is syntactically similar to a method
- Have no return type, not even **void**

```
class Box {
double width;
double height;
double depth;
// This is the constructor for Box.
```

```
Box() {
System.out.println("Constructing Box");
width = 10;
height = 10;
depth = 10;
}
// compute and return volume
double volume() {
return width * height * depth;
} }
class BoxDemo6 {
public static void main(String args[]) {
// declare, allocate, and initialize Box objects
Box mybox1 = new Box();
Box mybox2 = new Box();
double vol;
// get volume of first box
vol = mybox1.volume();
System.out.println("Volume is " + vol);
// get volume of second box
vol = mybox2.volume();
System.out.println("Volume is " + vol);
} }
```

Consider

Box mybox1 = new Box();

- **new Box( )** is calling the **Box( )** constructor
- When you do not explicitly define a constructor for a class, then Java creates a default constructor for the class
- The default constructor automatically initializes all instance variables to zero

Parameterized Constructors

```
class Box {
double width;
double height;
double depth;
// This is the constructor for Box.
Box(double w, double h, double d) {
width = w;
height = h;
depth = d;
}
// compute and return volume
double volume() {
return width * height * depth;
} }
class BoxDemo7 {
public static void main(String args[]) {
// declare, allocate, and initialize Box objects
Box mybox1 = new Box(10, 20, 15);
Box mybox2 = new Box(3, 6, 9);
```

```
double vol;
// get volume of first box
vol = mybox1.volume();
System.out.println("Volume is " + vol);
// get volume of second box
vol = mybox2.volume();
System.out.println("Volume is " + vol);
} }
```

The this Keyword

- ***this*** can be used inside any method to refer to the *current* object
- ***this*** is always a reference to the object on which the method was invoked

Example:

```
Box(double w, double h, double d) {
this.width = w;
this.height = h;
this.depth = d;
}
// Use this to resolve name-space collisions.
Box(double width, double height, double depth) {
this.width = width;
this.height = height;
this.depth = depth;
}
```

Garbage Collection

- In some languages, such as C++, dynamically allocated objects must be manually released by use of a **delete** operator
- Java takes a different approach; it handles deallocation for you automatically
- The technique that accomplishes this is called **garbage collection**
- When no references to an object exist, that object is assumed to be no longer needed, and the memory occupied by the object can be reclaimed
- There is no explicit need to destroy objects as in C++

The finalize( ) Method

- Sometimes an object will need to perform some action when it is destroyed
- For example, if an object is holding some non-Java resource such as a file handle or window character font, then you might want to make sure these resources are freed before an object is destroyed
- By using finalization, you can define specific actions that will occur when an object is just about to be reclaimed by the garbage collector
- To add a finalizer to a class, you simply define the **finalize( )** method
- Inside the **finalize( )** method you will specify those actions that must be performed before an object is destroyed

The **finalize( )** method has this general form:

```
protected void finalize( )
{
// finalization code here
}
```

- The keyword **protected** is a specifier that prevents access to **finalize( )** by code defined outside its class

- It is important to understand that **finalize( )** is only called just prior to garbage collection
- It is not called when an object goes out-of-scope, for example

- This means that you cannot know when—or even if—**finalize( )** will be executed
- Therefore, your program should provide other means of releasing system resources, etc., used by the object

Overloading Methods

- In Java it is possible to define two or more methods within the same class that share the same name, as long as their parameter declarations are different
- When an overloaded method is invoked, Java uses the type and/or number of arguments as its guide to determine which version of the overloaded method to actually call
- The return type alone is insufficient to distinguish two versions of a method

```
class OverloadDemo {
void test() {
System.out.println("No parameters");
}
// Overload test for one integer parameter.
void test(int a) {
System.out.println("a: " + a);
}
// Overload test for two integer parameters.
void test(int a, int b) {
System.out.println("a and b: " + a + " " + b);
}
// overload test for a double parameter
double test(double a) {
System.out.println("double a: " + a);
return a*a;
} }
class Overload {
public static void main(String args[]) {
OverloadDemo ob = new OverloadDemo();
double result;
// call all versions of test()
ob.test();
ob.test(10);
ob.test(10, 20);
result = ob.test(123.25);
System.out.println("Result of ob.test(123.25): " + result);
} }
```

- In some cases Java's automatic type conversions can play a role in overload resolution
- Java will employ its automatic type conversions only if no exact match is found

```
class OverloadDemo {
void test() {
System.out.println("No parameters");
}
void test(int a, int b) {
System.out.println("a and b: " + a + " " + b);
}
void test(double a) {
```

```
System.out.println("Inside test(double) a: " + a);
} }
class Overload {
public static void main(String args[]) {
OverloadDemo ob = new OverloadDemo();
int i = 88;
ob.test();
ob.test(10, 20);
ob.test(i); // this will invoke test(double)
ob.test(123.2); // this will invoke test(double)
} }
```

Overloading Constructors

Refer slides for example

Using Objects as Parameters

```
class Test {
int a, b;
Test(int i, int j) {
a = i;
b = j;
}
// return true if o is equal to the invoking object
boolean equals(Test o) {
if(o.a == a && o.b == b) return true;
else return false;
} }
class PassOb {
public static void main(String args[]) {
Test ob1 = new Test(100, 22);
Test ob2 = new Test(100, 22);
Test ob3 = new Test(-1, -1);
System.out.println("ob1 == ob2: " + ob1.equals(ob2));
System.out.println("ob1 == ob3: " + ob1.equals(ob3));
} }
```

- There are two ways that a computer language can pass an argument to a subroutine
- Call by value
- Call by reference
- when you pass a simple type to a method, it is passed by value

```
class Test {
void meth(int i, int j) {
i *= 2;
j /= 2;
} }
class CallByValue {
public static void main(String args[]) {
Test ob = new Test();
int a = 15, b = 20;
System.out.println("a and b before call: " + a + " " + b);
ob.meth(a, b);
System.out.println("a and b after call: " + a + " " + b);
```

```
} }
```
The output from this program is shown here:
```
a and b before call: 15 20
a and b after call: 15 20
```
- Objects are passed by reference
- Changes to the object inside the method *do* affect the object used as an argument

```
class Test {
int a, b;
Test(int i, int j) {
a = i;
b = j;
}
// pass an object
void meth(Test o) {
o.a *= 2;
o.b /= 2;
} }
class CallByRef {
public static void main(String args[]) {
Test ob = new Test(15, 20);
System.out.println("ob.a and ob.b before call: " +
ob.a + " " + ob.b);
ob.meth(ob);
System.out.println("ob.a and ob.b after call: " +
ob.a + " " + ob.b);
} }
```
This program generates the following output:
```
ob.a and ob.b before call: 15 20
ob.a and ob.b after call: 30 10
```
Returning Objects

- A method can return any type of data, including class types that you create

```
class Test {
int a;
Test(int i) {
a = i;
}
Test incrByTen() {
Test temp = new Test(a+10);
return temp;
} }
class RetOb {
public static void main(String args[]) {
Test ob1 = new Test(2);
Test ob2;
ob2 = ob1.incrByTen();
System.out.println("ob1.a: " + ob1.a);
System.out.println("ob2.a: " + ob2.a);
ob2 = ob2.incrByTen();
```

```
System.out.println("ob2.a after second increase: “ + ob2.a);
} }
```
The output generated by this program is shown here:
```
ob1.a: 2
ob2.a: 12
ob2.a after second increase: 22
```
Recursion

- Java supports *recursion*
- Recursion is the process of defining something in terms of itself
- As it relates to Java programming, recursion is the attribute that allows a method to call itself
- A method that calls itself is said to be *recursive*

Introducing Access Control

- Java’s access specifiers are **public**, **private**, and **protected**
- **protected** applies only when inheritance is involved
- When a member of a class is modified by the **public** specifier, then that member can be accessed by any other code
- When a member of a class is specified as **private**, then that member can only be accessed by other members of its class

Understanding static

- When a member is declared **static**, it can be accessed before any objects of its class are created, and without reference to any object
- The most common example of a **static** member is **main( )**
- **main( )** is declared as **static** because it must be called before any objects exist
- Instance variables declared as **static** are, essentially, global variables

Methods declared as static have several restrictions:

- They can only call other **static** methods
- They must only access **static** data
- They cannot refer to **this** or **super** in any way
- We can declare a **static** block which gets executed exactly once, when the class is first loaded

```
class UseStatic {
static int a = 3;
static int b;
static void meth(int x) {
System.out.println("x = " + x);
System.out.println("a = " + a);
System.out.println("b = " + b);
}
static {
System.out.println("Static block initialized.");
b = a * 4;
}
public static void main(String args[]) {
meth(42);
} }
```

- As soon as the **UseStatic** class is loaded, all of the **static** statements are run

- First, **a** is set to **3**, then the **static** block executes (printing a message), and finally, **b** is initialized to **a * 4** or **12**
- Then **main( )** is called, which calls **meth( )**, passing **42** to **x**
- If you wish to call a **static** method from outside its class, you can do so using the following general form:

  ***classname.method*( )**

  - Here, *classname* is the name of the class in which the **static** method is declared

```
class StaticDemo {
static int a = 42;
static int b = 99;
static void callme() {
System.out.println("a = " + a);
} }
class StaticByName {
public static void main(String args[]) {
StaticDemo.callme();
System.out.println("b = " + StaticDemo.b);
} }
```

Here is the output of this program:

```
a = 42
b = 99
```

Introducing final

- A variable can be declared as **final**
- Doing so prevents its contents from being modified
- We must initialize a **final** variable when it is declared
- final int FILE_NEW = 1;
- final int FILE_OPEN = 2;

Introducing Nested and Inner Classes

- It is possible to define a class within another class
- The scope of a nested class is bounded by the scope of its enclosing class
- If class B is defined within class A, then B is known to A, but not outside of A
- A nested class has access to the members, including private members, of the class in which it is nested
- However, the enclosing class does not have access to the members of the nested class
- There are two types of nested classes: ***static*** and ***non-static***
- A static nested class is one which has the **static** modifier applied
- The most important type of nested class is the *inner* class
- An inner class is a non-static nested class
- It has access to all of the variables and methods of its outer class

```
class Outer {
int outer_x = 100;
void test() {
Inner inner = new Inner();
inner.display();
}
// this is an inner class
class Inner {
void display() {
```

```
System.out.println("display: outer_x = " + outer_x);
} } }
class InnerClassDemo {
public static void main(String args[]) {
Outer outer = new Outer();
outer.test();
} }
```

- It is important to realize that class **Inner** is known only within the scope of class **Outer**
- The Java compiler generates an error message if any code outside of class **Outer** attempts to instantiate class **Inner**

Using Command-Line Arguments

```
class CommandLine {
public static void main(String args[]) {
for(int i=0; i<args.length; i++)
System.out.println("args[" + i + "]: " +
args[i]);
} }
```

Execution:

java CommandLine this is a test 100 -1

When you do, you will see the following output:

```
args[0]: this     args[1]: is     args[2]: a     args[3]: test
args[4]: 100      args[5]: -1
```

Inheritance

- Allows the creation of hierarchical classifications
- In the terminology of Java, a class that is inherited is called a ***superclass***
- The class that does the inheriting is called a ***subclass***
- A subclass is a specialized version of a superclass
- It inherits all of the instance variables and methods defined by the superclass and adds its own, unique elements
- Incorporate the definition of one class into another by using the **extends** keyword

```
// Create a superclass.
class A {
int i, j;
void showij() {
System.out.println("i and j: " + i + " " + j);
} }
// Create a subclass by extending class A.
class B extends A {
int k;
void showk() {
System.out.println("k: " + k);
}
void sum() {
System.out.println("i+j+k: " + (i+j+k));
} }
class SimpleInheritance {
public static void main(String args[]) {
A superOb = new A();
```

```
B subOb = new B();
// The superclass may be used by itself.
superOb.i = 10;
superOb.j = 20;
System.out.println("Contents of superOb: ");
superOb.showij();
System.out.println();
/* The subclass has access to all public members of its superclass. */
subOb.i = 7;
subOb.j = 8;
subOb.k = 9;
System.out.println("Contents of subOb: ");
subOb.showij();
subOb.showk();
System.out.println();
System.out.println("Sum of i, j and k in subOb:");
subOb.sum();
} }
```

- A subclass can be a superclass for another subclass
- The general form of a **class** declaration that inherits a superclass is shown here:

class subclass-name extends superclass-name {
// body of class
}

- ***Java does not support the inheritance of multiple superclasses into a single subclass***

Member Access and Inheritance

- Subclass cannot access those members of the superclass that have been declared as **private**

```
// Create a superclass.
class A {
int i; // public by default
private int j; // private to A
void setij(int x, int y) {
i = x;
j = y;
} }
// A's j is not accessible here.
class B extends A {
int total;
void sum() {
total = i + j; // ERROR, j is not accessible here
} }
```

Using super

- Whenever a subclass needs to refer to its immediate superclass, it can do so by use of the keyword **super**
- **super** has two general forms
- The first calls the superclass' constructor
- The second is used to access a member of the superclass that has been hidden by a member of a subclass
- A subclass can call a constructor method defined by its superclass by use of the following form of **super**:

super(parameter-list);

- *parameter-list* specifies any parameters needed by the constructor in the superclass

```
// BoxWeight now uses super to initialize its Box attributes.
class BoxWeight extends Box {
double weight; // weight of box
// initialize width, height, and depth using super()
BoxWeight(double w, double h, double d, double m) {
super(w, h, d); // call superclass constructor
weight = m;
} }
```

- This leaves **Box** free to make these values **private** if desired

A Second Use for super

- It always refers to the superclass of the subclass in which it is used
- This usage has the following general form: **super.*member***
- Here, *member* can be either a method or an instance variable
- This second form of **super** is most applicable to situations in which member names of a subclass hide members by the same name in the superclass

```
// Using super to overcome name hiding.
class A {
int i;
}
// Create a subclass by extending class A.
class B extends A {
int i; // this i hides the i in A
B(int a, int b) {
super.i = a; // i in A
i = b; // i in B
}
void show() {
System.out.println("i in superclass: " + super.i);
System.out.println("i in subclass: " + i);
} }

class UseSuper {
public static void main(String args[]) {
B subOb = new B(1, 2);
subOb.show();
} }
```

This program displays the following:

```
i in superclass: 1
i in subclass: 2
```